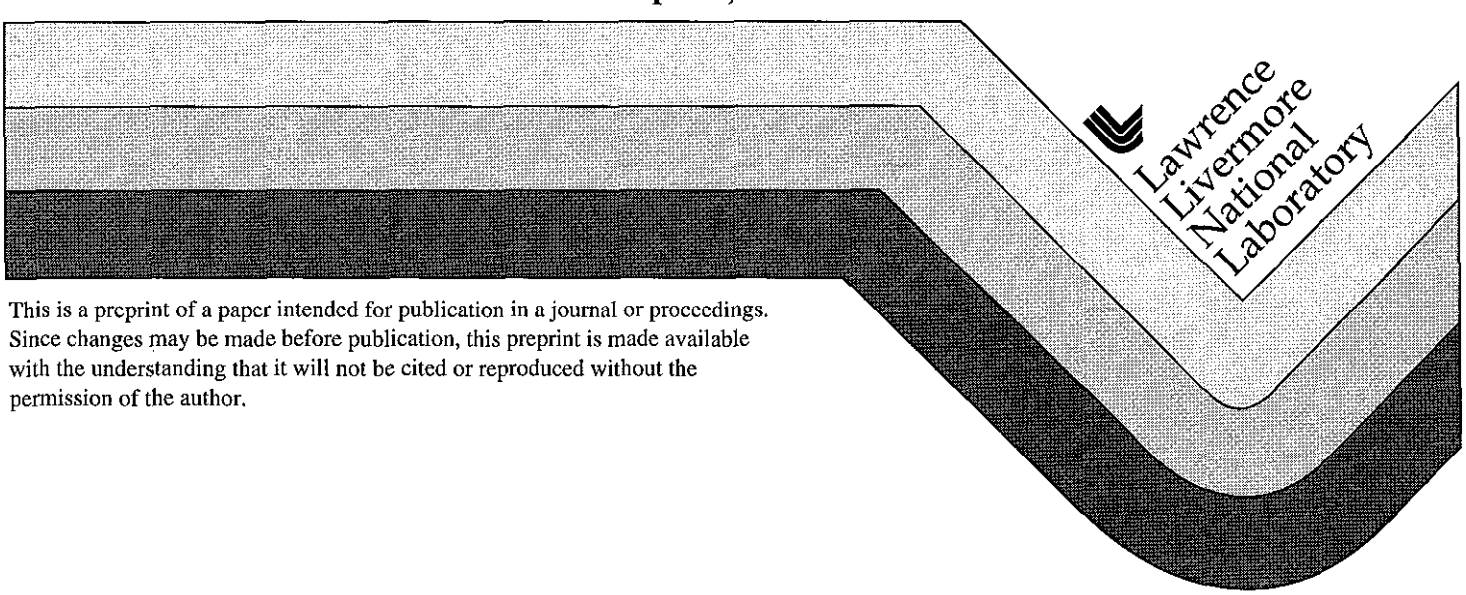
UCRL-JC-130363
PREPRINT

Visualizing Hilbert Curves

Nelson Max

This paper was prepared for submittal to the
Institute of Electrical and Electronics Engineers Visualization '98
Research Triangle Park, NC
October 18-23, 1998

April 1, 1998



Visualizing Hilbert Curves

Nelson Max
Lawrence Livermore National Laboratory
P.O. Box 808 / L-307
Livermore, California 94551, USA
(max2@llnl.gov)

Abstract

A computer animated movie was produced, illustrating both 2D and 3D Hilbert curves, and showing the transition from 2D to 3D with the help of volume rendering.

Introduction

Hilbert curves are continuous curves which pass at least once through each point of a square or cube. They can be defined as the limit of a sequence of mappings of successively smaller dyadic subintervals of the unit interval to small subsquares or subcubes.

These finite approximations are useful for coding images or volumes. Area coherence in an image produces high correlation in the sequence of data values at pixels, when they are traversed in the order of a Hilbert curve approximation, allowing efficient data compression. Similarly, tracing out a volume Hilbert curve can take advantage of data coherence in all three dimensions.

This paper describes the production of an animated film illustrating 2D and 3D Hilbert curves. The content of the film is described in section 1, the mathematical definitions and algorithms for the approximations are given in section 2, and the volume rendering techniques are discussed in section 3.

1. The Film Plot

The animation starts with a circular tube, which deforms continuously through smooth piecewise-circular approximations to the 2D Hilbert curve. The fourth order approximation is shown in figure 1. (See color supplement at the end of the volume for figures 1 through 8.) The tube changes to a square cross section, and by the fifth order approximation, shown in figure 2, it is touching itself. The circular arcs then square off so that the approximation appears to cover the square, and the surfaces become partially transparent, to reveal the glowing volume density shown in figure 3.

The curve is yellow, with its parametrization indicated by short segments in light green, purple, and dark green, repeated in eight cycles. The origin of the parametrization is advanced along the curve as the animation progresses, so that it looks like a crawling colored snake. This indicates the path of the curve even when the whole square is filled up. Volume rendering extends this color indication of the path to three dimensions, and the marble tile background helps in perceiving the volume opacity.

The closed curve breaks between a purple and dark green band and the front part begins filling up the cube as a 3D Hilbert curve. By figure 4, the second purple band has started into the cube, and by figure 5, the whole curve has moved over to the cube, and again becomes closed.

Figures 3 through 5 show high order approximations, using a recursive rendering algorithm described in section 3 which only subdivides squares or cubes which are not of homogeneous color. Between any two frames in the animation, many tiny squares or cubes change color, so the motion is jerky. Figure 6 shows a switch to the third order approximation, where the color boundaries move smoothly. The surface is incised between cubes which are not adjacent on the approximation path, and in figure 7, the incisions have widened to reveal the squared off tube. This is the reverse of the process, not illustrated here, between figures 2 and 3. Finally in figure 8 the tube becomes rounded, to give the 3D version of figure 1. In the animation sequence including figures 5 through 8 the camera is slowly rotated about the curve, so that motion parallax can help make the 3D structure more evident.

2. Defining the Approximations.

The 2D Hilbert curve maps a t in the unit interval [0, 1] to a pair (x, y) in the unit square [0, 1] x [0, 1]. An approximation of order n assigns the first n binary digits of x and y, given the first $2n$ binary digits of t. The first bits of x and

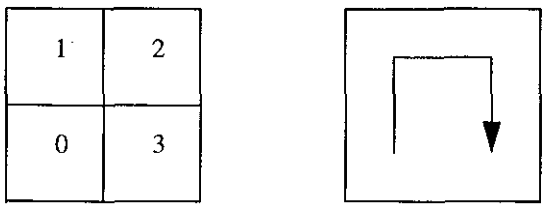


Figure 9. Ordering of the four subsquares.

y come from the first two bits of t according to the ordering of the four subsquares shown in figure 9. Thus

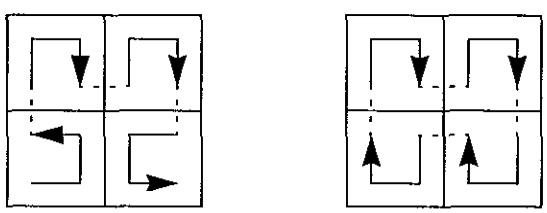
xbit2D[4] = {0, 0, 0, 1},

ybit2D[4] = {0, 1, 1, 0},

and, for the inverse mapping,

tbits2D[2][2] = {0, 1, 3, 2}.

The higher order bits are defined recursively, using rotated versions of this order, as shown at the left of figure 10. In the squares labeled 1 and 2 in figure 9, the pattern is



a) Hilbert's curve. b) My variant.

Figure 10. Second order approximations.

just a reduced and translated version of the pattern in figure 9, but in squares 0 and 4, it is reflected and rotated. Apart from the scaling and translation, the basic transformation for the pattern is given by one of the following four matrices:

$$R2D[0] = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, R2D[1] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, R2D[2] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and}$$

$$R2D[3] = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}.$$

I assume that for the nth approximation, the coordinates of x and y are given by n bit integers, with the highest order bit of x indicated by x_0, the next highest by x_1, and so on. When divided by the implied denominator 2^n, these x and y coordinates give the position of the lower left corner of the corresponding subsquare. Similarly, the $2n$ bit integer for t is divided by 2^{2n} to get the left hand endpoint of the corresponding subinterval. Thus the arrays xbit and ybit determine x_0 and y_0 from the first two bits t_{01} of t. To get the next highest bits x_1 and y_1 from the next two bits t_{23} of t, we must translate figure 9 by (-.5, -.5) so that the center is at the origin, rotate by R2D[t_{01}], and then translate back. The product of these three affine transformations can be represented by a 3 x 3 integer matrix for homogeneous coordinates, which is used to transform the vector (xbit2D[t_{23}], ybit2D[t_{23}], 1) to get (x_1, y_1, 1). However it is more efficient to code, and easier to understand, if the 2 x 2 matrices given above are used, and the two translations are done separately. As show in the code below, the coordinates are multiplied by two before the first translation, and divided by two after the second, to keep all arithmetic in integers.

```
void t_to_xy(int n, int t, int *x, int*y) {
  int j,k,rt[2][2], rq[2][2], va[2], vb[2];
  identity_matrix2D(rt);
  *x = *y = 0;
  for (k = n-1; k >= 0; ++k) {
    j = 3 & (t >> (2*k));
    va[0] = 2*xbit2D[j] - 1;
    va[1] = 2*ybit2D[j] - 1;
    matrix_times_vector2D(rt, va, vb);
    *x += ((vb[0] + 1)/2) << k;
    *y += ((vb[1] + 1)/2) << k;
    if (k > 0) {
      matrix_copy2D(rt, rq);
      if (k == n-1)
        matrix_multiply2D(rq, S2D[j], rt);
      else
        matrix_multiply2D(rq, R2D[j], rt);}
  }}
```

In the next stage we rotate figure 10a) by the matrix R[t_{01}], shrink it to half size, and use it each of the four subsquares. Thus the combined rotation R2D[t_{01}]R2D[t_{23}] is used to determine x_2 and y_2 from t_{45}. We continue recursively generating the subsequent digits. The inverse function determining t from x and y is defined similarly using tbits2D.

For the film plot described above, I needed a closed curve, so I used the variant second approximation shown in figure 10b). To move the pattern in figure 9 to the four subsquares in figure 10b), I used the matrices:

$$S2D[0] = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, S2D[1] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, S2D[2] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and}$$

$$S2D[3] = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}.$$

To guarantee the continuity of the limit curve, I then used the matrices R2D for all subsequent bits of x and y, so that a half-sized copy of figure 10a) appears in each of the subsquares of figure 10b), oriented by the matrix S2D[t_{01}]. Thus the combined rotation to determine x_2 and y_2 from t_{45} is S2D[t_{01}] R2D[t_{23}], and the R2D matrices are also used in the rest of the recursion. The resulting curve agrees with Hilbert's on the top half of the square, and the bottom half is the mirror image of the top half.

The Hilbert curve in3D is generated in the same way, using 3D vectors and matrices. The basic pattern shown in figure 11 results in

xbit3D[8] = {0, 0, 0, 0, 1, 1, 1, 1},

ybit3D[8] = {0, 1, 1, 0, 0, 1, 1, 0},

zbit3D[8] = {0, 0, 1, 1, 1, 1, 0, 0}, and

tbits[2][2][2] = {0, 3, 1, 2, 7, 4, 6, 5}.

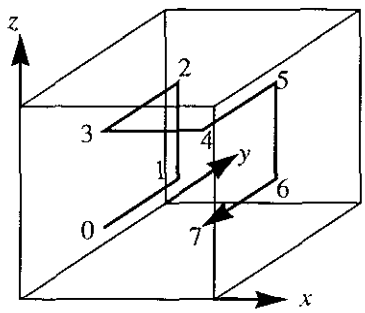


Figure 11. Order of the eight subcubes.

The matrices to rotate this pattern into position in each of the eight subcubes are essentially those given in [Liu97] but I have incorporated a reflection that reverses the order of traversal where necessary. This obviates keeping track of the traversal order, which makes the algorithm in [Liu97] unnecessarily complicated. Thus the 3D rotation matrices are:

$$R3D[0] = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R3D[1] = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad R3D[2] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$R3D[3] = \begin{bmatrix} 0 & 0 & -1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad R3D[4] = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

$$R3D[5] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R3D[6] = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \text{ and } R3D[7] = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and the recursion is similar to the 2D case.

As in the 2D case, I needed a set of revised matrices for the second order approximation, to produce a close curve, which agrees with the 3D Hilbert curve of [Liu97] in the top half of the cube, and follows its mirror image in the bottom half.

To get the piecewise circular approximation to the curve inside the square (x, y) shown in figure 12, I computed t from (x, y) and then found the previous square (xp, yp) from t-1, and the following square (xf, yf) from t+1. I connected the center A of the edge separating square (x, y) and square (xp, yp) to the center B of the edge separating square (x, y) and square (xf, yf), by a 90° circular arc whose center C is the corner where these three squares meet. If the three squares are in a row, with no common corner, I joined A and B by a straight line. The same scheme works in 3D; the circular arc joins the centers of two faces of the cube.

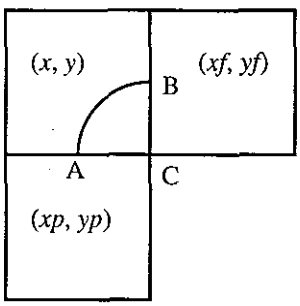


Figure 12. Construction of a circular arc.

It was a challenge to deform this arc through a continuous family of piecewise circular curves into its position in the next approximation. I had initially planned to end the film by unwinding the 3D Hilbert curve back to a simple circle, but I did not have time to design the necessary 3D deformation. It was also a challenge to get a smoothly varying parametrization of the piecewise circular curves in the family, for the purpose of deciding where to switch between the different colored bands. The parametrization had to be nearly proportional to arc length, but also assign the same lengths to the circular arc in figure 12 and the straight line in the case that all three squares are in a row.

3. Rendering Issues.

As described in section 1, and shown in the color figures 1 through 8, the colored bands move along the curve during the animation. Thus, if f is a real number proportional to the frame count, and s is the real parameter in [0,1] along the curve, the color is determined by a periodic piecewise constant function $C(s - f)$ of the difference $s - f$. The colors have four components, red, green, blue, and opacity α, which is interpreted as the extinction coefficient for volume rendering.

In the limit curves, which fill in the whole square or cube, the path of the curve is indicated by these moving colored bands. They are revealed in the 3D case using semi-transparent volume rendering.

The volume rendering is based on recursive octree subdivision of the level 0 unit cube. The octree node cube at level k is indexed by integers (x, y, z) obtained by multiplying the coordinates of its front lower left corner by 2^k. This cube corresponds to an integer t by the inverse mapping

discussed in section 2, and to the subinterval of length 2^{-3k} of the unit interval, starting at $s = 2^{-3k}t$. If $[s - f, s + 2^{-3k} - f]$ lies within one of the ranges where the color C is constant, or if $k = kmax$, the maximum level of recursion, the cube has a constant color, and can be composited onto the image. If not, it is subdivided into its eight subcubes, they are sorted in back to front order according to the position of the viewpoint, and the recursive routine is called for each.

The volume projection uses the SGI hardware pipeline, as described in [WMS98]. The projections of the edges of a cube divide the image plane into polygons onto which a single front cube face and a single rear cube face project. The thickness d varies linearly across such a polygon, and can be interpolated as a texture coordinate by the texture mapping hardware. The second texture coordinate is the extinction coefficient τ, which in this application is constant on each cube. The texture map stores $\alpha = 1. - \exp(-d\tau)$, which is the polygon opacity needed for correct compositing of a semitransparent volume density. (See [WMS97] for details.)

When I produced the volume rendered Hilbert curve with this rendering method, it looked too fuzzy, because there were no clearly defined surfaces bounding each volume region. Therefore I added semitransparent polygons to the front surfaces of a cube, if such surfaces separated volumes of different colors. The determination of these separating faces, for each cube rendered by the volume recursion above, also requires recursion. The recursive pseudocode below is called for each front-facing face of a volume rendered cube C at recursion level k.

```
Draw (k, F, C){
  if(F lies on one of the outside faces of
    the level 0 cube) {render(F); return;}
  find the cube D of the same size as C on
    the opposite side of face F;
  if(Color(D) is homogeneous or k == kmax)
    if(Color(D) != Color(C) ) render(F);
  else
    subdivide F into four subfaces Fs[],
      bounding four subcubes Cs[]of C;
    for(i = 0; i < 4; ++i)
      Draw(k+1, Fs[i], Cs[i]);
  }
```

Because all rendering was done by the SGI hardware, through OpenGL calls, it only took a few seconds a frame. The images were generated at 1270 X 970 pixel resolution, and then averaged down to 635 X 485 resolution for video recording, providing 2 X 2 supersampled antialiasing.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48. Brett Keating produced the marble texture, using algorithms from [Perl85]. Mark Duchaineau provided a window opening and image read-back facility that relieved me of learning the details of X. Dietmar Saupe pointed me to reference [Liu97]. Brian Cabral and Dan Schikore provided debugging help. Jan Nunes, Ross Gaunt, and Eugene Cronshagen recorded the video. Don Patterson, Mike Miller, and Charlene Frey connected my system rapidly back up to our internal network after a hacker break-in, so that I was able to record the video and print the paper.

References

[Liu97] Xian Liu and Günther Schrack, "An algorithm for encoding and decoding the 3-D Hilbert order", IEEE Transactions on Image Processing Vol 6, No. 9 (1997) pp. 1333 - 1337.

[Perl85] Ken Perlin, "An image synthesizer", Computer Graphics Vol. 19 No 3 (Siggraph '85 Proceedings, 1985) pp. 287 - 296.

[WMS98] Peter Williams, Nelson Max, and Clifford Stein, "A high accuracy volume renderer for unstructured data", IEEE Transactions on Visualization and Computer Graphics, Vol. 4 No. 1 (1998)

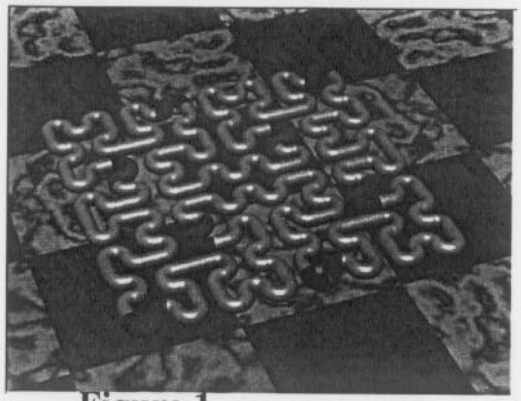


Figure 1.

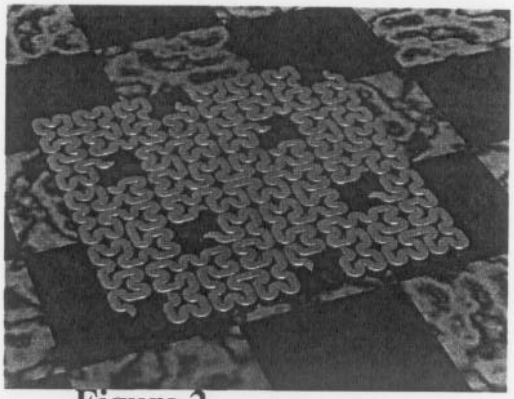


Figure 2.

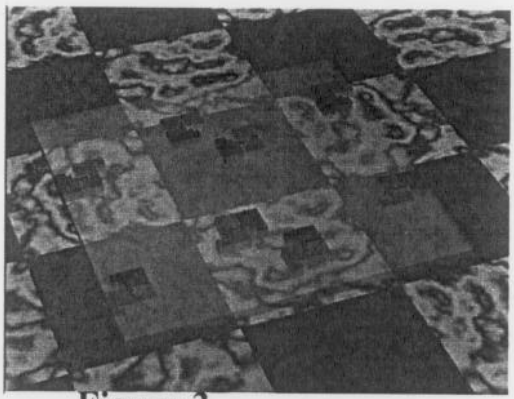


Figure 3.

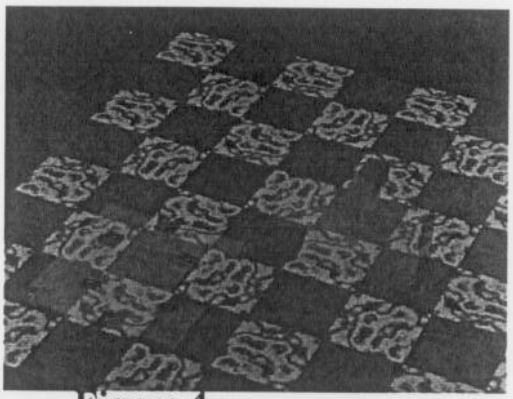


Figure 4.

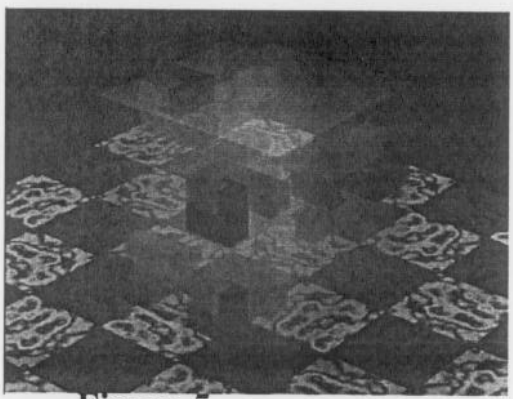


Figure 5.

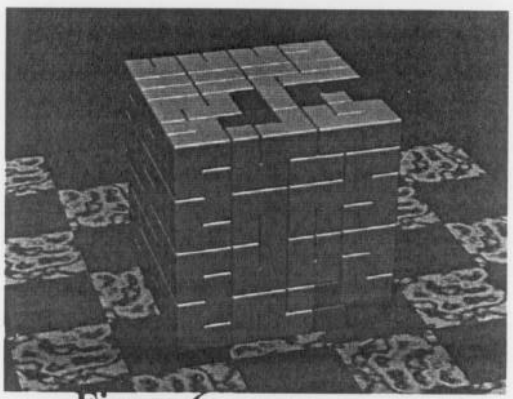


Figure 6.

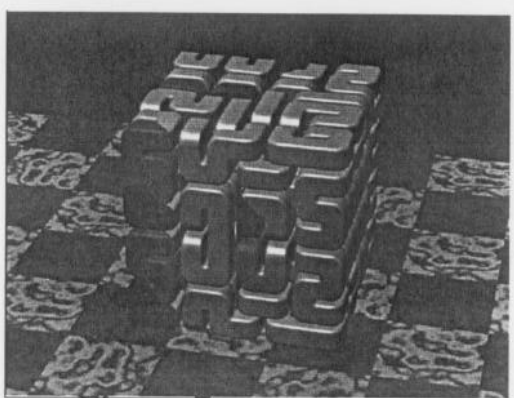


Figure 7.

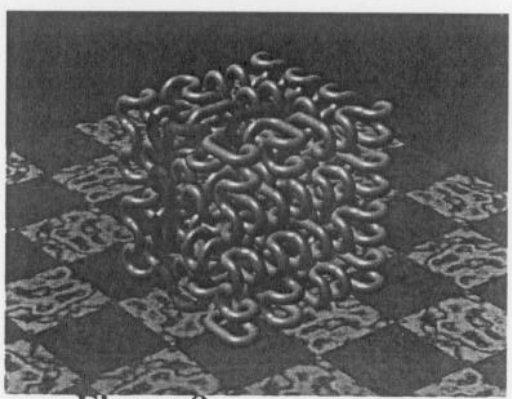


Figure 8.